Using Kernel Extensions to Decrease the Latency of User-Level Communication Primitives

University of New Mexico Technical Report Number CS96-4

Rolf Riesen

Ph.D. proposal submitted to the Computer Science Department of the University of New Mexico

December 10, 1996

Abstract

CPUs, network interfaces, and networks are improving, providing higher bandwidths and lower latencies. System software overhead makes it impossible for a user application to achieve bandwidths and latencies near the hardware limits. This is especially true for remote handler invocation. Typically, the remote node has to trap into the kernel and perform an expensive context switch to the handler. This hampers global communication operations and runtime systems, such as the one for Cilk and Split-C for example.

Executing the untrusted remote handler inside the operating system kernel eliminates the overhead of context switches and disrupted cashes. Several methods to execute untrusted user code in a privileged environment exist. The research proposed in this paper compares these methods and attempts to prove that a kernel embedded interpreter has the necessary performance and safety characteristics to be the ideal method for remote handler invocation in massively parallel systems.

Contents

Chapter 1

Introduction

A statement of the problem and why it should be solved [55].

Explicit message passing and various forms of one-sided communications (puts and gets) are efficient methods to harness the power of a distributed memory machine. Much effort and research is being spent on improving message passing latency and bandwidth for massively parallel systems. While this research is important, it does not address remote execution. Often it is necessary to perform a simple action, such as adding two numbers or incrementing a counter, when a message arrives. Most mechanisms that support remote execution have an overhead of ten to a hundred times the cost to deliver a simple message.

Collective communications, such as broadcast and global sum, need low latencies to send messages between nodes. On each node, however, it is important that the message be processed quickly and then sent on to other nodes. This is difficult for routines that allow user-defined operations. While common operations, such as global sum and global max, are often part of the native message passing system, user specified functions have to be executed in user space and the overhead of a context switch, to run the function, delays the global operation.

Another class of user-level systems that require remote execution of user (or library) specified code, are runtime systems. For example, Cilk's [9] runtime system achieves load balancing through work-stealing. An idle node with no work chooses another node at random and sends it a work-steal request. The "victim" node should respond quickly with new work, or a negative acknowledgment. The goals are to minimize the impact on working nodes and to get an idle node working again as quickly as possible.

Another example is Split-C [16], which uses remote put and get operations to transfer data from one node to another. To avoid deadlocks, allow for synchronization, and guarantee atomicity, small handlers that are part of the Split-C runtime system, have to be executed on most message arrivals. Split-C is usually implemented on top of an active message layer. The arrival of an active message triggers the execution of a handler that performs a small amount of work, such as incrementing a counter, and then sends a reply. It is crucial that this handler be invoked as soon as possible after the active message arrives.

In all three examples (collective operations, Cilk, and Split-C), the user application or a third party runtime system specifies a function to be executed when a message arrives. Response time can be reduced if these functions are invoked immediately when a message arrives. Several methods have been devised to address this need.

Active messages [100] transmit the address of a function to be executed on the remote node, along with a small amount of data passed as parameters to the function. Most implementations poll for incoming messages and then jump to the address specified in the message header. Handler invocation can be very fast. Single digit microsecond latencies are reported in the literature. Latencies are much higher if the remote node is busy with a long computation and is not polling at the moment the active message arrives.

Intel's NX message passing system for the Paragon [73, 74] provides the functions `hrecv()` and `hsend()` to execute user defined handlers on message arrival or completion of a send. The implementation is interrupt driven. After calling `hsend()`, the application continues to run. When the send completes, that is, the data has been delivered to the remote node, the sending application is interrupted and the handler specified as a parameter to `hsend()` is run. Similarly, an `hrecv()` sets up a buffer and matching criteria for an incoming message and specifies a handler. When a message is deposited in the buffer, the receiving application is interrupted and the handler is run. The overhead to context switch to the handler is high compared to the cost of receiving a message (about 300μs versus 14μs).

In the Puma operating system [104, 105, 84] a portal event handler can be attached to any Puma portal. The user specified handler is run after the message has been deposited in the portal. As with `hsend()` and `hrecv()`, this requires a relatively expensive context switch.

As these examples illustrate, methods that provide the necessary functionality exist. All of them have a performance impact on the receiving node and introduce significant delays in the propagation of messages to other nodes.

Consider the example of a broadcast where a single node sends information to all other nodes in the application. In a distributed memory architecture this can be done using a fanout tree. The originating node sends a message to one of its neighbors. The neighbor then passes the message to one of its other neighbors, while the originating node is copying the message to yet another node. This pattern continues until all nodes have received the message.

Each node has to receive the message and then send it on to the appropriate nodes in the fanout tree. Implementing a broadcast using basic point-to-point message passing operations has several drawbacks. If a node in the middle of the fanout tree has not completed its current task, its participation in the fanout will be delayed and the children of that node will have to wait, even if they are ready to receive the broadcast data. On architectures where the network interface can only be accessed in supervisor mode, the necessary trap into the kernel and back to user level further increases the cost for each message receipt and send.

Using (non-polling) active messages, Intel's `hsend()` and `hrecv()`, or Puma portal event handlers, the problem of delaying a broadcast by a busy intermediate node can be avoided. There is a cost associated with this. When a message arrives, a context switch from the currently running application to the the user specified handler and back to the application occurs. These interrupt driven context switches are expensive, especially on modern RISC CPUs which have to save and restore a large amount of internal context. Context switches disrupt cache and TLB contents and impact the currently running application.

The research proposed in this document explores ways to avoid these additional context switches and thereby improve the performance of runtime systems and other communication primitives that require user specified handlers.

More specifically, we will explore methods to execute untrusted user code in supervisor mode, while the kernel is receiving a message. If all desired user specified functions were known a priori, they could be built directly into the operating system kernel. On message arrival the kernel would then simply execute that function and, after that, return to the user level. On systems with a general purpose message processor, such as the Intel Paragon, the coprocessor could execute the handler without interrupting the user code that is running on the main CPU.

Of course, it is impossible to anticipate all possible user handlers. Therefore, an approach to let user applications insert code into the kernel at runtime is needed. Since this code is untrusted and executed in supervisor mode at the time a message arrives, precautions must be taken to ensure the integrity of the kernel and other applications.

In the next chapter we will look in detail at the cost of user level message handlers and the potential savings of running handlers in kernel mode. We will also look at methods to safely execute untrusted code in the kernel of an operating system. In Chapter 3 we discuss work that is related to this proposal. In Chapter 4 we characterize our proposed solution and establish the criteria to measure and compare our solution. Finally, in Chapter 5, we outline a statement of work and a schedule.

Chapter 2

Key Ideas and Concepts

The candidate's ideas and insights for solving the problem and any preliminary results he may have obtained [55].

Based on the examples listed in the introduction, we define a common model that describes the execution path on a node. During a broadcast, each node receives a message that is to be distributed to its children in the fanout tree. (The originating node is at the root of the tree and generates the message.) Each node identifies its children when the broadcast message is received and then sends a copy of the message to each child. Therefore, for a broadcast a node needs to be able to receive a message, calculate the addresses of its children in the fanout tree, and send messages to them.

For a global sum, a node receives data from its children, sums it, adds its own contribution, and sends the result to the parent. Instead of sum, the operation can be any function, even one specified by the user.

The Cilk work-stealing example is, in principle, no different than the one above. A node receives a work-steal request, checks its task queue, and sends a message back to the originator. Checking the task queue, potentially removing an entry from the queue and sending a work order back, is slightly more complicated than determining the children in a fanout tree. Nevertheless, the basic principle remains the same. A node receives one or more messages, does some processing, and then, potentially, sends one or more messages. The routine performing the processing is called a handler.

Most remote operations in the Split-C runtime system require the update of a counter. A node receives a message, increments or decrements a counter and sends some data or an acknowledgment back to the originating node.

We should note that this work concentrates on efficient and quick processing of external, asynchronous events. Another approach is to have the user program or the runtime system occasionally check for new messages (polling). If a message has arrived, the corresponding handler is called. This approach works especially well on architectures where the network interface is mapped into the user's address space.

The routine that polls for new messages can read the message and call the handler specified in the message. Assuming that the polling operation occurs with sufficient frequency, this avoids costly interrupts and allows for rapid dispatching of handlers.

On architectures where incoming messages generate interrupts and calls to the operating system kernel are necessary to send messages, the main advantage of polling disappears. Furthermore, for operations such as a broadcast or work-stealing where the currently running thread does not directly contribute (it is doing something else now), the response time is better and much more predictable, when an interrupt driven implementation is used.

There is another disadvantage of a polling implementation. Sometimes it is impossible to poll frequently enough. For example, during a long computation in a BLAS library call, the application writer cannot insert polling operations.

In Section 2.1 we will concentrate on a model that is common to the above examples and many others in daily use on high-performance systems. This section should convince us that executing a user defined handler in the kernel has performance advantages. In Section 2.2 we will look at different methods to execute untrusted code safely inside an operating system kernel.

2.1 Cost Model

Instead of examining each of the earlier examples in more detail, we will now concentrate on a common model and study the events and associated costs on a given node. Figure 2.1 depicts the principle graphically. The arrival of one or more messages triggers a small amount of computation that results in zero or more messages being sent.

We now consider the events that take place on a node from the time a message arrives until the handler completes and normal operation on the node is resumed. There are three cases.

1. Architectures that require supervisor (kernel mode) privileges to access the network interface.

2. Architectures that map the network interface into the user's address space.

3. Architectures that dedicate a general purpose CPU to the handling of communication.

For each architecture we consider the cost of a handler running in user mode, including the necessary context switches, and compare it with the case where the handler is executed in the kernel itself.

2.1.1 Restricted Access to Network Interface

On systems of this type it is necessary to trap into the kernel to send or receive a message. The network interface uses physical addresses to access main memory

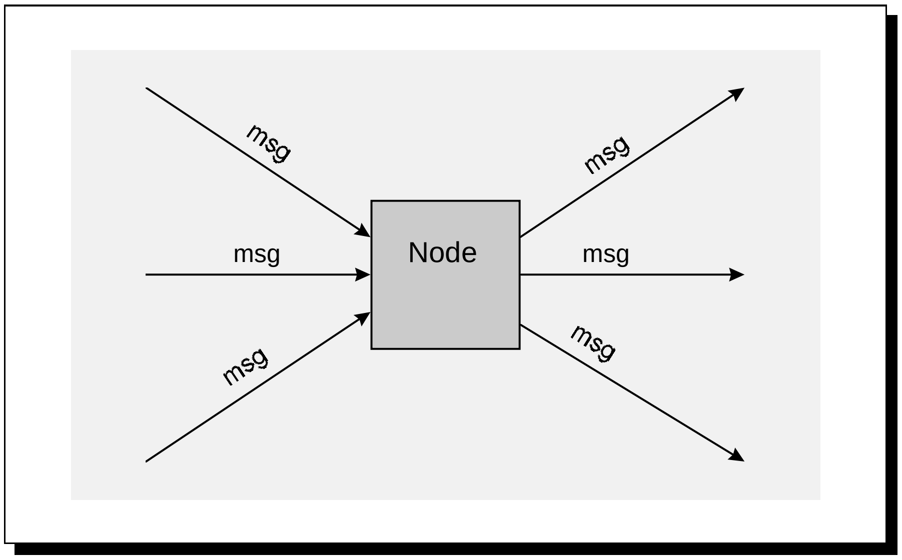

Figure 2.1: **Common Events**: A node receives one or more messages, does some processing, and then, potentially, sends one or more messages.

on the node[1]. This simplifies the design of the network interface, but requires access in supervisor mode to prevent user applications from destroying or reading portions of main memory that must be protected. Protecting the network interface is also necessary to prevent sending of data to arbitrary nodes in the system. This architectural model is the most common. The Intel Paragon, even though it has two and three CPUs on each node, can be operated in this manner under the Puma operating system.

Handler at User Level

Figure 2.2 shows the events that take place. We are interested in reducing the total time the user application is delayed: Δ_{delay}. Also of interest is the response time (Δ_{resp}); the time from message arrival until the handler begins to run.

Table 2.1 summarizes the time intervals that characterize the events on a given node. A message arrives at a node and causes an interrupt. The CPU switches into supervisor mode, saves the current context, and sets up the context for the kernel to run. This time interval is symbolized by Δ_{int}. The kernel then proceeds to read the message from the network interface into a buffer in main memory. We represent this time interval with Δ_{rcv}. The value for Δ_{rcv} varies with the length of

[1]Some architectures, such as the iPSC/860, do not let the network interface access memory directly. Instead, the network interface consists of a send and a receive FIFO. A program has to read and write the FIFO for each word transferred. Access to the FIFO registers has to be controlled to prevent applications from reading data that is intended for another process.

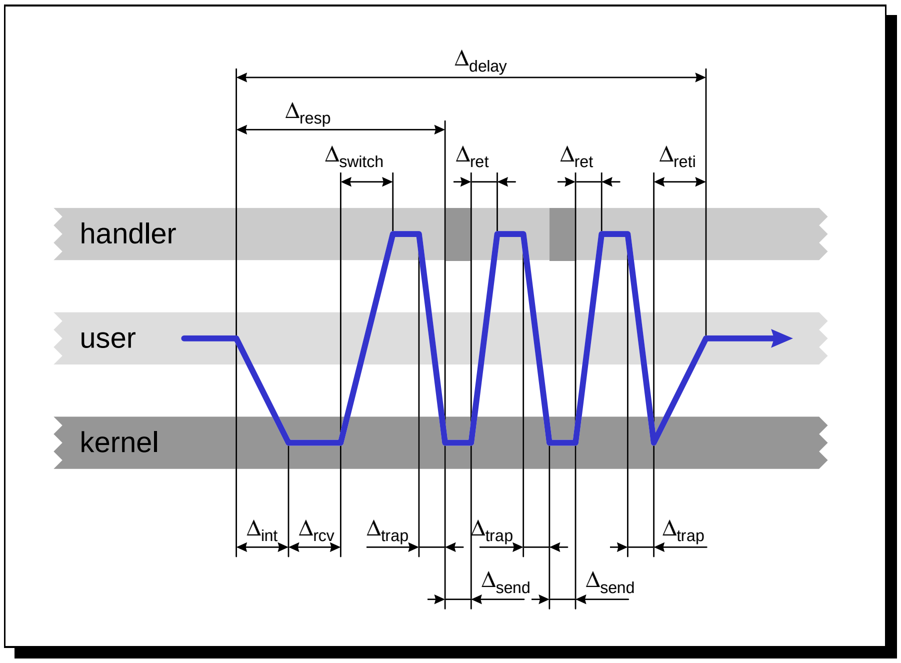

Figure 2.2: **Execution Flow on a Node with Restricted Access to the Network Interface**: An arriving message interrupts the currently running user application. The kernel, in supervisor mode, receives the message and deposits it into a buffer. A context switch to the appropriate handler occurs. To send messages, the handler has to trap into the kernel. A final trap into the kernel is necessary to resume the interrupted user application.

the message. Furthermore, for longer messages it is possible to engage the DMA unit and continue processing while the message is being received. For our analysis we ignore this optimization and assume the CPU remains busy during the receipt of the message.

After determining which handler to invoke, it takes Δ_{switch} time to switch context and start running the handler. In the example of Figure 2.2 the handler does some processing and then sends two messages. For each message the handler has to trap into the kernel, Δ_{trap}. The kernel will send the message (with the same assumption about staying busy as for message reception). This takes Δ_{send} time, which increases with larger messages. The time to return from the kernel trap back to the handler is Δ_{ret}.

When the handler finishes, it traps into the kernel one more time to re-establish the original context. The time it takes to return from an interrupt to the user level process is Δ_{reti}.

Table 2.1: Time Intervals of Interest

Symbol	Description
Δ_{int}	Time to switch context and enter kernel mode on a interrupt
Δ_{rcv}	Time to read a message from the network
Δ_{switch}	Time to instantiate and switch to a user level handler
Δ_{trap}	Time to trap into the kernel
Δ_{send}	Time to inject a message into the network
Δ_{ret}	Time to return from a trap
Δ_{reti}	Time to return from an interrupt
Δ_{delay}	Total time a user application is delayed
Δ_{resp}	Shortest possible response time for a handler

Handler in Kernel

We now examine the same example under the assumption that the handler can be executed while in kernel mode. Figure 2.3 shows the timing diagram.

As before, the currently running user application is interrupted when a message arrives. The node switches context at a cost of Δ_{int} and enters kernel mode. It takes Δ_{rcv} time to receive the message into main memory. Now, however, executing the handler is a simple and cheap function call. The handler performs the same operations as before. When it sends a message, it simply calls the appropriate function in the kernel.

This method saves the time of a context switch to the handler, a trap into the kernel to restore the originally running process, and the two traps and returns to send the messages:

$$\Delta_{saved} = \Delta_{switch} + \Delta_{trap} + 2(\Delta_{trap} + \Delta_{ret})$$

We will see in Section 2.2 that safely executing a user defined handler in the kernel costs additional time. It is the goal of this work to establish how much overhead is imposed by the various techniques to execute handlers in kernel mode and compare that to Δ_{saved} and other benefits of this approach.

The saving of two traps and returns, $2(\Delta_{trap} + \Delta_{ret})$, is specific to our example. For handlers that require more traps into the kernel, the time savings would further increase. For handlers with fewer kernel requests, the time savings could decrease to as little as

$$\Delta_{saved} = \Delta_{switch} + \Delta_{trap}$$

Most handlers require at least one system call to send a reply.

2.1.2 Mapped Network Interface

Systems such as the Thinking Machine CM-5 and the Meiko CS-2 have the capability of mapping the network interface into the user address space. This mapping requires

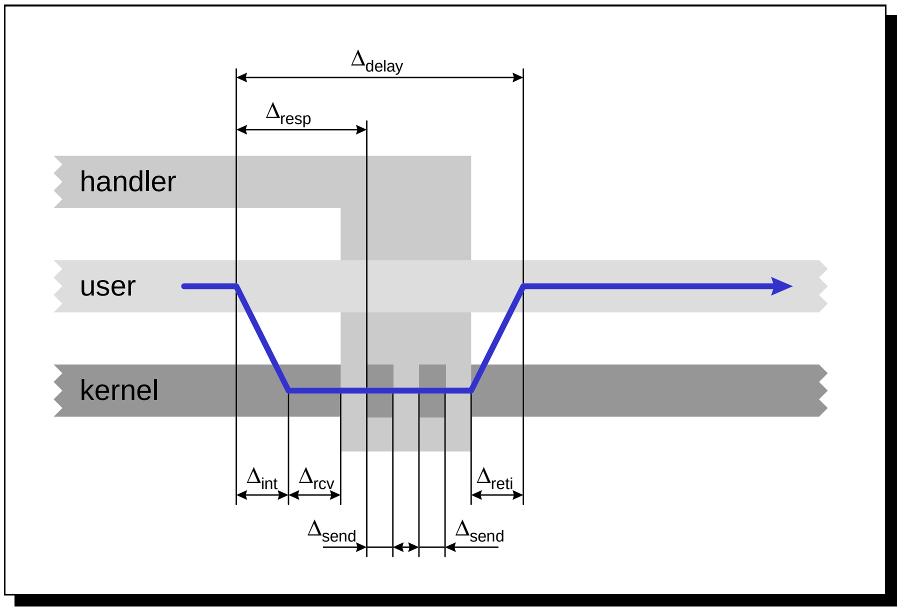

Figure 2.3: **Restricted Access to the Network Interface, Handler in the Kernel**: An arriving message interrupts the currently running user application. The kernel, in supervisor mode, receives the message and deposits it into a buffer. Immediately thereafter, the handler starts running. To send messages, the handler simply calls the appropriate kernel function. A return from interrupt resumes execution of the user application.

that the interface accept and interpret application relative virtual addresses or that the interface represents the endpoint of a virtual channel (FIFO) that can be read from or written to, a single word at a time. The advantage is that messages can be received and sent by user level applications with no expensive traps into the kernel.

Handler at User Level

Figure 2.4 shows the flow of execution under these circumstances. While polling the network interface on an architecture that allows user access to the interface is the simplest way to low latency, we assume that an incoming message interrupts the currently running process. This interrupt is necessary to guarantee response time and corresponds to the two other architectures described in this section.

An incoming message interrupts the currently running process. The kernel invokes a generic handler that reads the message into a buffer. The handler runs in non-privileged mode and accesses the network interface directly. Depending on the message content, the generic handler then calls the appropriate specific handler. This is a simple function call and does not require the saving and restoring of a

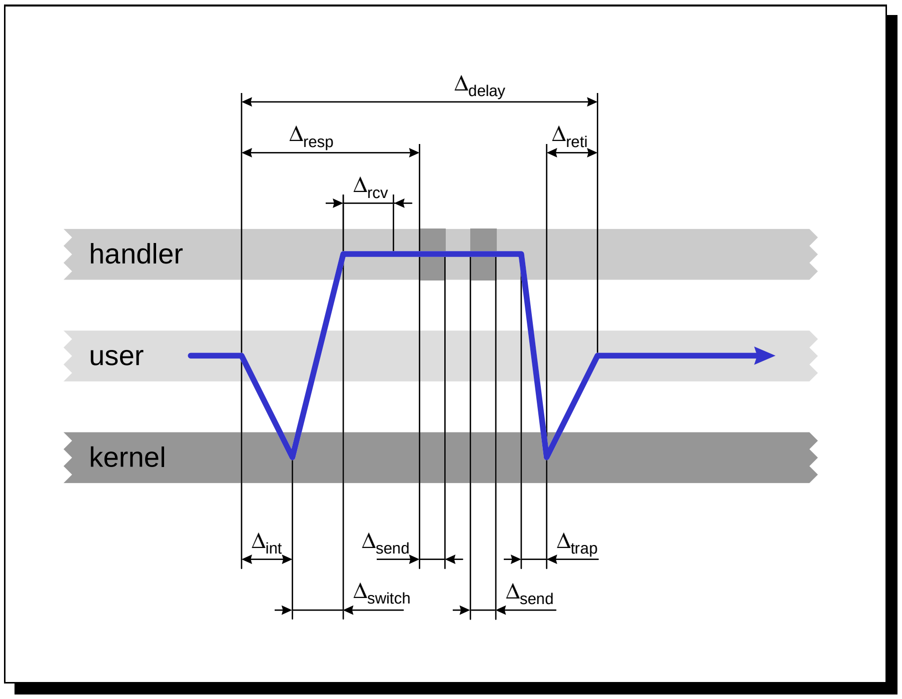

Figure 2.4: **Mapped Network Interface**: An arriving message interrupts the currently running application to activate the handler. The handler can receive and send messages without trapping into the kernel. A single trap is necessary at the completion of the handler to return execution control to the interrupted application.

context.

The handler is the same as in the previous examples, except that no traps into the kernel are necessary to send messages. A single trap at the end is needed to restore the originally running process.

Handler in Kernel

Figure 2.5 shows the situation where the handler is executed in the kernel. Again, the arriving message triggers an interrupt and forces control flow into the kernel. The kernel receives the message into a buffer and then simply calls the appropriate handler. The handler does its processing and sends its two messages. A return from interrupt leads back into user mode where the interrupted application resumes.

Since the network interface is accessible from user mode, no savings are possible by executing the handler in the kernel. However, a context switch to the handler and a trap to restore the original context can be avoided. The saved time is:

$$\Delta_{saved} = \Delta_{switch} + \Delta_{trap}$$

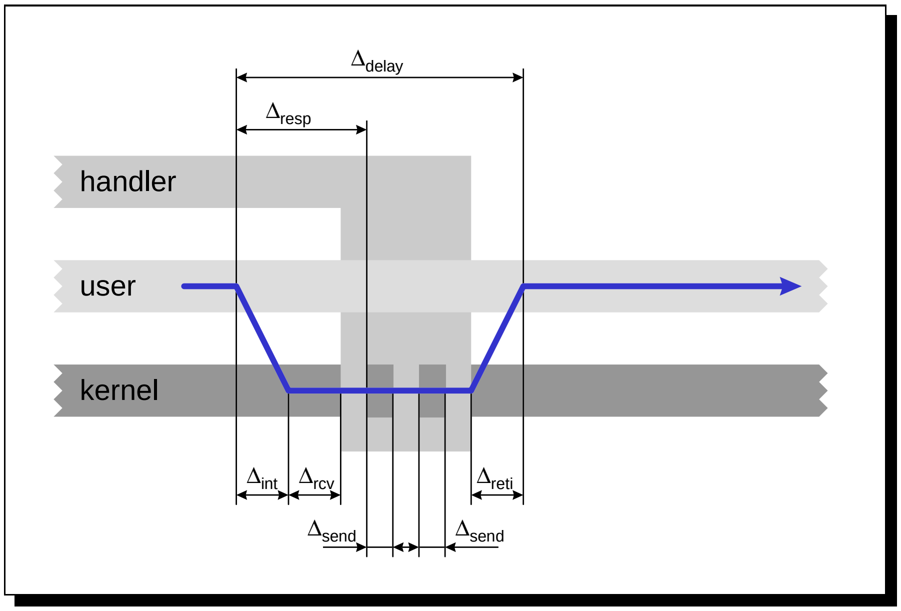

Figure 2.5: **Mapped Network Interface, Handler in Kernel**: An arriving message interrupts the currently running process. The kernel receives the message into a buffer in main memory and calls the handler without a context switch. The handler executes and sends its two messages. A return from interrupt resumes execution of the original user application.

2.1.3 General Purpose CPU as a Message Coprocessor

The Intel Paragon as well as the Pentium Pro based Teraflop computer to be installed at Sandia National Laboratories have the ability to dedicate a general purpose CPU to the task of sending and receiving messages. The CPUs share the memory bus with the DMA units and the network interface. A snooping cache coherency protocol ensures the integrity of data in the caches and main memory. In the message coprocessor mode, one CPU remains at the user level executing application code, while the second CPU remains in kernel mode polling the network interface. The two CPUs use a mailbox in shared memory to exchange information. For this to be effective, the second CPU cannot execute code in user mode. For our purposes, we let the second CPU interrupt the first one when it is time to run a handler. Figure 2.6 shows the execution flow.

Handler at User Level

The polling CPU detects the arrival of a message and receives it into main memory. Then it sends an interrupt signal to the user CPU to force a context switch. The user CPU executes the handler and, at the end, traps into the kernel to restore the original context. To send a message, a handler on the user CPU deposits a send request in a shared mailbox. The message coprocessor uses that information to perform the actual send. Therefore, the processing performed by one CPU and the message sending performed by the second CPU can overlap.

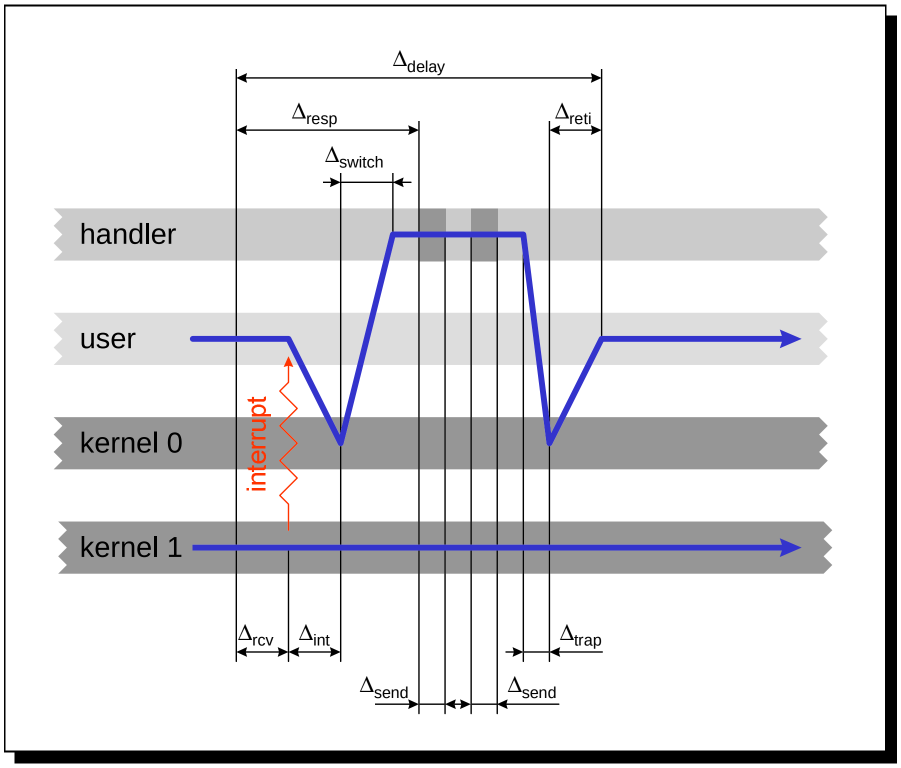

Figure 2.6: **General Purpose CPU**: An arriving message is read into a buffer by the CPU dedicated to message transfer. It then interrupts the CPU running the user application so it can perform a context switch to the user level handler. Messages between CPUs are sent by exchanging information through a mailbox in shared memory. The dedicated CPU performs the actual send. A trap on the first CPU is necessary to restore the original context.

Handler in Kernel

Figure 2.7 illustrates the situation with a dedicated CPU performing message passing as well as executing the handler. At no time is the currently running process on the user CPU interrupted! Without considering the time savings derived from running the handler on a separate CPU, we get the following result:

$$\Delta_{saved} = \Delta_{int} + \Delta_{switch} + \Delta_{trap} + \Delta_{reti}$$

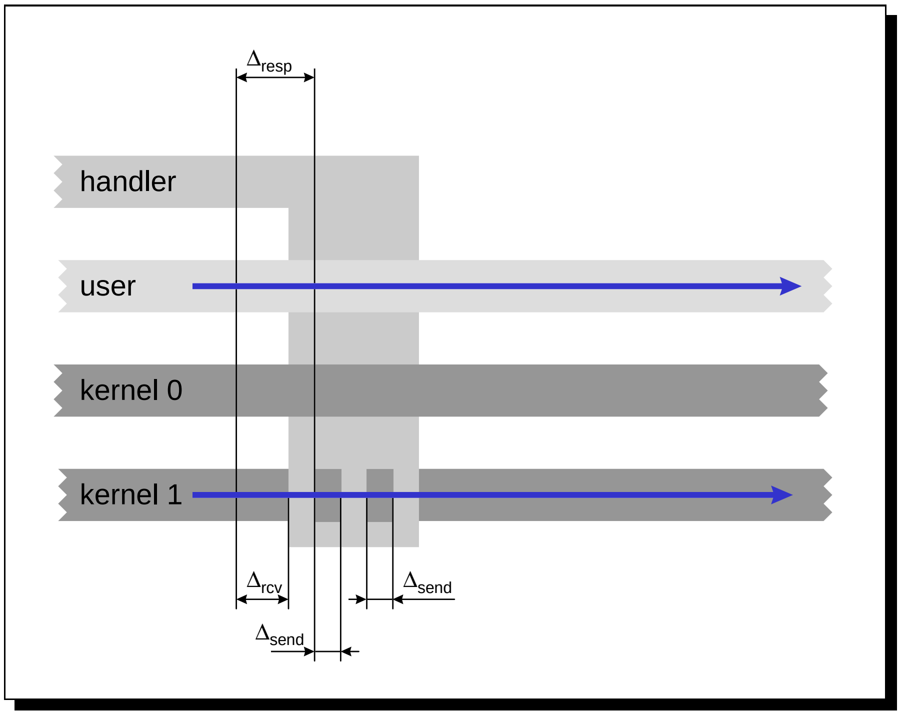

Figure 2.7: **General Purpose CPU, Handler in Kernel**: An arriving message is read into a buffer by the CPU dedicated to message transfer. It then executes the handler. The user process running on the other CPU is never interrupted!

2.1.4 Analysis

In all three architectural models a saving of at least $\Delta_{saved} = \Delta_{switch} + \Delta_{trap}$ is possible. Context switch and trap times on a RISC CPU such as the Intel i860 used in the Intel Paragon, can be considerable. Saving and restoring the floating-point pipeline state is especially costly, bringing the total cost to several tens of micro seconds.

We anticipate that most handlers are short and will execute quickly. In these cases, some overhead to safely execute a handler in the kernel can be easily tolerated. However, if the overhead of executing a handler in the kernel becomes larger than Δ_{saved}, it would be better to run the handler at user level.

Figure 2.8 illustrates the relationship between a handler in the kernel and one at the user level. The handler at the user level pays an up-front cost of at least $\Delta_{startup}$ before it starts executing. (Δ_{saved} varies with the handler and the method used to execute a handler in the kernel. $\Delta_{startup}$ is constant for a given operating system.) A handler running in supervisor mode has to be prevented from gaining privileges that it would not have if it ran in user mode. The methods used to prevent handlers from gaining privileges, impose an overhead and make the kernel handler execute more slowly. The differences in execution speeds are represented by the different slopes of the two curves. There is a crossover point after which the overhead of executing the handler in the kernel exceeds the time taken by the user level handler. Therefore, if the amount of work a handler has to do exceeds a certain threshold, then it is better to run the handler in user mode.

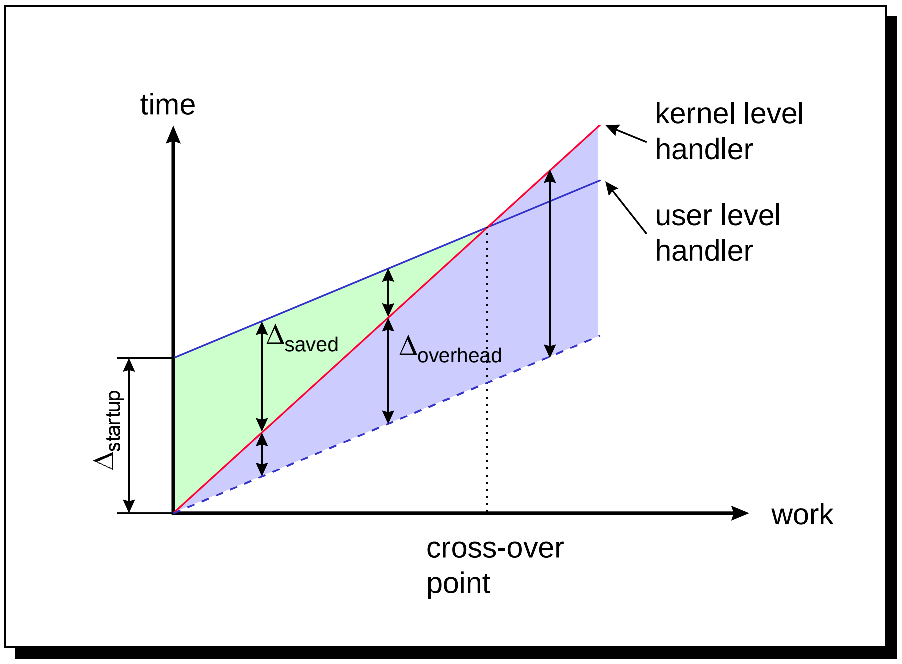

Figure 2.8: **User Level and Kernel Level Handlers**: A handler in the kernel runs slower because of the overhead to ensure integrity of the system. A user-level handler pays a cost up front ($\Delta_{startup}$) but then executes faster because no memory access checks are needed for example. The overhead to run in the kernel accumulates until it becomes larger than $\Delta_{startup}$. At that point a handler should be run at user level.

$\Delta_{startup}$ is determined by the hardware and the operating system. The overhead of running a handler in the kernel is determined by the method used to ensure the integrity of the system. In the next section we will look at various methods to run handlers in the kernel. We find that each method has its own overhead, dependent on the amount of work (number of instructions) and the type of work (load and store versus computation) a handler performs.

In Figure 2.9 we give a hypothetical example of how the various methods of running a handler in the kernel compare to running the handler at user level. Since the overhead characteristics for each method are dependent on the instruction mix of the handler, an analysis would have to be done for each handler of interest. The varying slopes of each step of each curve represent memory accesses and CPU internal instructions. The cost for the two types of operations are different in each method. Memory access patterns are also different, giving each method a unique execution profile for a given handler. Since the amount of work a given handler performs is bounded, it will be possible to determine the method that executes a given handler the fastest.

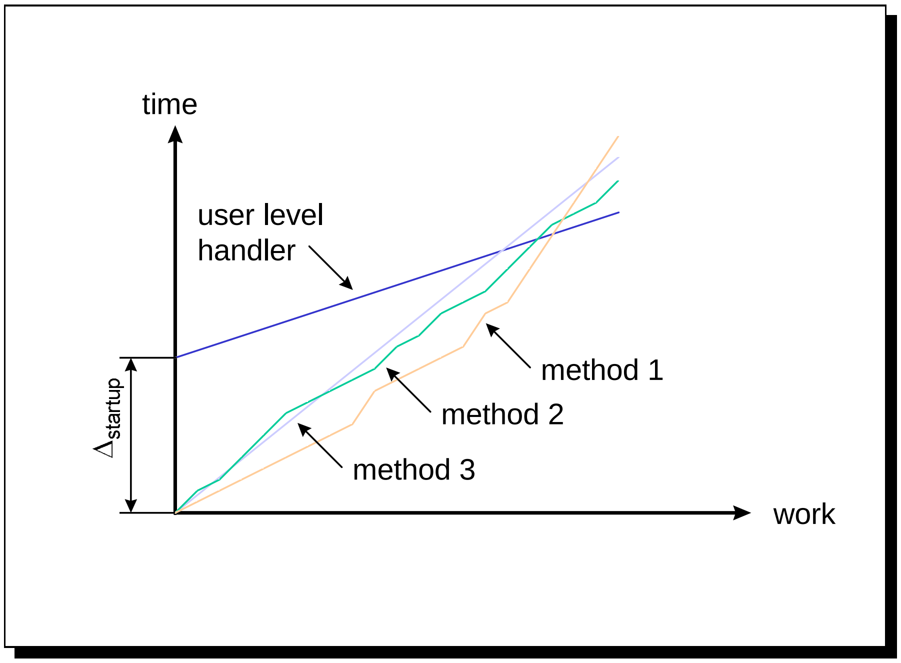

Figure 2.9: **Example of Kernel Handlers**: Depending on the method chosen and the particular handler, the time it takes to do a given amount of work will change. Choose the method that requires the least time for a given handler. $\Delta_{startup}$ is the cost to get a handler started in user space.

The class of handlers described in the examples of Section 1 share two common characteristics: they are short and use simple operations. It is possible that for this class of handlers one of the methods described in the next section executes fastest most of the time. We will perform measurements under the Puma operating system running on an Intel Paragon and the DOE Teraflop system to characterize the execution profile of this class of handlers and find the method that is best suited for these handlers.

2.2 Kernel Embedded Handlers

The idea of executing user code (a handler) in kernel mode is not new. In Chapter 3 we will discuss previous work in this area as well as other approaches to avoid costly context switches. In this section we discuss four software approaches for user level handlers in the kernel.

All four methods deal with the problem of protecting the integrity of the system from user code that is running inside the kernel with supervisor mode privileges. Specifically, a handler running inside the kernel must not be able to access memory or memory mapped devices that it cannot access when run in user mode. Furthermore, privileged instructions that cannot be executed in user mode must not be executed by kernel embedded handlers. The running time of a handler also needs to be restricted to limit processor resource use on behalf of a user process. We call these restrictions *privilege restrictions*. The methods described in the following sections either enforce these privilege restrictions, or ensure that a handler voluntarily follows them.

Common to all methods are the following steps. At *code insertion* time a user level handler is inserted into the kernel and associated with an event. When the event occurs, the handler is executed inside the kernel without a context switch to user mode. During code insertion the kernel may perform a method-specific *inspection* and *link* step. The code is then stored inside the kernel where it cannot be altered by the user. For each method, we describe the steps performed during code insertion and how the privilege restrictions are enforced or guaranteed during execution.

The first method, software based fault isolation, comes in two flavors: sandboxing and segment matching. Sandboxing forces all memory accesses to specific memory segments called the sandbox[2]. Segment matching checks each memory access and reports an error if an attempt to access memory outside the sandbox is detected.

The second approach is to trust a compiler to produce code that does not misuse any of the privileges gained by running in supervisor mode. Code produced by the trusted compiler is assumed to not violate the privilege restrictions. Software based fault isolation and the trusted compiler approach have attracted much attention recently with the advent of extensible operating systems such as the MIT Exo kernel [28] and SPIN [7, 8].

A third approach, interpretation, is currently out of fashion based on the expectation that interpretation is too slow [86]. On the other hand, interpretation is enjoying a comeback in the form of Java, a byte code interpreted language. Java is used to create applets that are sent across the World Wide Web (WWW) [63, 36]. Although not executed in kernel mode, applets cannot be trusted since origin and content are unknown at the time of execution. Applets can access files on the client, connect to other hosts, and send mail with the same privileges as the user who down-

[2]As Charlie Crowley points out, the term sanboxing is misleading to a parent. A playpen is used to keep children within a given space. A sandbox does that only when the children cooperate.

loaded the applet. This ability poses a safety hazard. Java bytecode is interpreted in software to protect against such unwanted activities of an applet.

The fourth possible approach to embed a handler inside a kernel is code inspection. If the kernel can formally verify that a given function does not violate the restrictions imposed on inserted code, then the kernel can execute that function safely.

2.2.1 Software Based Fault Isolation

Sandboxing, the first form of software based fault isolation, derives its name from the idea of isolating a user program in a sandbox where it can execute safely without the possibility of damaging anything outside the sandbox. Like hardware implemented memory protection, sandboxing ensures that unsafe instructions cannot access memory outside the sandbox. The idea is presented by Wahbe et al. [102], where the authors consider write and jump instructions as unsafe. However, as Small and Seltzer [86] point out, read operations must also be regarded as unsafe, since some hardware devices change state when they are read. Of course, privileged instructions such as `reset`, instructions that change the memory access privileges, instructions that disable (the watchdog timer) interrupts, and illegal instructions must be prohibited. Sandboxed code has to be inspected during code insertion. There is no need to trust the compiler to have done the sandboxing correctly.

A sandbox consists of two memory segments. One for text and the other for data. The segments are aligned such that the upper n bits, the *segment index*, of all addresses in each segment are the same. During compilation, code is inserted before each unsafe instruction to prevent access to locations outside the allocated segments. This code sequence forces the upper n bits of the unsafe instruction address to be the same as the segment index. This prevents any reads and writes outside the data segment and prevents jumps to locations outside the text segment.

Figure 2.10 and Figure 2.11 illustrate examples of sandboxing a load and a jump instruction. The Intel i860 instruction to the left is sandboxed by the instruction sequence to the right. Five dedicated registers are used. Two are needed for load and stores (r_{loc}, r_{data_seg}), two for jumps (r_{target}, r_{code_seg}), and one can be shared by all sandboxed instructions (r_{mask}). r_{mask} can be shared if the segment index is of the same width for both the data and the code segment. The inspection that is performed during code insertion has to ensure that the dedicated registers are not modified outside the sandboxing sequence.

Segment matching is the second form of software based fault isolation. It is an extension of sandboxing and allows pinpointing the location of an offending instruction and simplifies debugging. Instead of simply forcing the upper n bits to be the same as the segment index, the n bits are compared with the segment index. If they are the same, it is safe to execute the instruction. If not, execution flow branches to an error handling routine that can output detailed information and abort the offending procedure. Figures 2.12 and 2.13 illustrate segment matching.

`ld.l` *off*`(`r_i`),` r_d
    // off *is a register or a*
    // 16 bit address offset

(a) Normal Code

`addu` *off* `,` r_i`,` r_{loc}
    // Calc location (must not
    // be in a branch delay slot)
`and` r_{loc}`,` r_{mask}`,` r_{loc}
    // Clear segment index
`or` r_{loc}`,` r_{data_seg}`,` r_{loc}
    // Set data segment index
`ld.l 0(`r_{loc}`),` r_d
    // Use sandboxed address

(b) Sandboxed Code

Figure 2.10: **Sandboxing a Load Instruction in i860 Assembly**: r_{mask} , r_{loc}, and r_{data_seg} are dedicated registers the user code cannot use.

`bri` r_i

(a) Normal Code

`and` r_{mask}`,` r_i`,` r_{target}
    // Clear segment index
`or` r_{target}`,` r_{code_seg}`,` r_{target}
    // Set code segment index
`bri` r_{target}
    // Use sandboxed address

(b) Sandboxed Code

Figure 2.11: **Sandboxing a Jump Instruction in i860 Assembly**: r_{mask} , r_{target}, and r_{code_seg} are dedicated registers the user code cannot use.

To prevent self-modifying code, two segments are needed. One for data (static, heap, and stack), and one for the code. Load and store instructions are restricted to the data segment, while branch instructions have to stay in the code segment. Note that branches into any part of the sandboxing code are possible and the sandboxing code has to be written in such a manner that none of the dedicated registers can be compromised. Both software based fault isolation techniques require five dedicated registers. (Under certain conditions it is possible to save one register in the segment matching case.) On a RISC CPU with 32 general purpose registers, this is not a big problem. On a CISC architecture, such as the Pentium Pro, with only 8 general purpose registers, the techniques can still be used, albeit at the cost of slower performance. In the case of the Pentium Pro which has segmentation registers, it would be interesting to compare the cost of using segment registers instead of software based fault isolation.

```
ld.l off(r_i), r_d
     // off is a register or a
     // 16 bit address offset
```

(a) Normal Code

```
addu off, r_i, r_loc
     // Calculate target address
and r_mask, r_loc, r_tmp
     // Retrieve segment index
xor r_tmp, r_data_seg
     // Comp with data seg index
bc error
     // report error
ld.l 0(r_loc), r_d
     // Safe: execute load (store)
```

(b) Segment Matching Code

Figure 2.12: **Load Instruction With Segment Matching in i860 Assembly**: $\mathtt{r}_{mask}$, $\mathtt{r}_{loc}$, and $\mathtt{r}_{data\_seg}$ are dedicated registers the user code cannot use. $\mathtt{r}_{tmp}$ is a temporary register that can be used outside the segment matching code.

The authors of [102] report an average fault isolation overhead of 4.3% in a variety of benchmarks, isolating writes and jumps only, and allowing reads from any location of mapped memory. This agrees with the reported 3% to 7% in [96].

The insertion of the sandboxing or segment matching instructions before each unsafe instruction can be done at compile time or at the time the code is inserted into the kernel. In the former case, the kernel has to ensure that the function has been properly sandboxed. Wahbe et al. present an algorithm to do that [102]. The algorithm ensures the following:

- All jumps, PC relative or absolute targets, are within the code segment.
- Register indirect jumps are sandboxed and use the dedicated register r_{target}.
- All direct memory accesses are to addresses within the data segment.
- Register indirect memory accesses are sandboxed and use the dedicated register r_{loc}.
- The handler does not contain privileged or illegal instructions.
- None of the dedicated registers (r_{loc}, r_{data_seg}, r_{target}, r_{code_seg}, r_{mask}) are updated outside the sandboxing codes.

To limit running time, a watchdog timer is used. The timer is set at the time the handler starts executing and terminates the handler if it goes off before the handler has finished.

```
bri r_i
```

(a) Normal Code

```
mov r_i, r_target
      // Move target into dedicated reg
and r_mask, r_target, r_tmp
      // Retrieve segment index
xor r_tmp, r_code_seg, r0
      // Comp with code seg index
bc error
      // report error
bri r_target
      // Safe: execute jump
```

(b) Segment Matching Code

Figure 2.13: **Branch Instruction With Segment Matching in i860 Assembly**: $\mathtt{r}_{mask}$, $\mathtt{r}_{target}$, and $\mathtt{r}_{code\_seg}$ are dedicated registers the user code cannot use. $\mathtt{r}_{tmp}$ is a temporary register that can be used outside the segment matching code.

2.2.2 Trusted Compiler

An approach that is currently being investigated as part of the SPIN project is to trust the compiler to generate code that obeys the privilege restrictions. There are two possibilities for a kernel to trust the compiler. If the compiler is a stand-alone tool, as is usually the case, it has to "sign" the generated binary in such a manner that the kernel can be sure that only the trusted compiler could have generated this particular binary. This requires code in the kernel to verify the signature.

The second method is to make the compiler part of the operating system. One problem with the trusted compiler approach is that that the amount of trusted code grows significantly. The size of the Puma kernel, for which we are considering kernel extensions, is about 75 kB, while the size of a typical compiler is usually measured in mega bytes. Furthermore, the runtime systems of high-level languages can be large. For performance reasons, the runtime system has to stay resident in physical memory taking up valuable space.

In some sense, all operating systems trust the compiler that was used to compile them. With the trusted compiler approach the operating system places an additional kind of trust on the compiler. The operating system now assumes that the compiler not only produces correct code, but also code that does not violate the privilege restrictions. It is up to the compiler to reject code that might access memory that is accessible in kernel mode but not user mode.

For this approach, code insertion consists of linking the code into the kernel and, depending on whether the compiler is part of the operating system or not, verifying the identity of the code producer.

2.2.3 Kernel Embedded Interpreter

Perhaps the simplest approach, at least conceptually, is interpretation of the user code. A kernel embedded interpreter can validate each instruction before executing it and easily guarantee that the privilege restrictions are obeyed. Time bounds can be enforced by simply counting the number of instructions executed (weights could be assigned to each instruction if execution times vary widely).

Interpretation has been successfully employed in a variety of situations [66, 61, 36], but is generally considered to be slow or only applicable if the input language can be sufficiently restricted. For the handlers anticipated, we believe that interpretation is ideal. The handlers are small and perform simple tasks. It should be possible to gather the most often used constructs and sequences into a virtual machine which can be optimized to execute efficiently [76, 78]. Then, using indirect threaded code or direct threaded code techniques, build an extremely fast interpreter [6, 21, 49, 31]. These techniques have been used to implement the Forth language [79]. The B (a predecessor to C) compiler for the PDP-7 generated threaded code [81] as did the Fortran IV compiler for the PDP-11 [6]. The object oriented language Actor is based on token threading [23]. QuickBasic 4.0 is based on a threaded P-code interpreter [101]. The handler code is threaded during code insertion.

It is one of the goals of this work to design a virtual machine that is general purpose, yet highly optimized to the interpretation of code that is produced when compiling handlers.

2.2.4 Code Inspection

Under certain circumstances it is possible to inspect a binary image and guarantee that it obeys the privilege restrictions. A control flow analysis must be performed to ensure that no branches outside the handler take place. In order to limit the run time, loops are required to have fixed starting and ending values and backward branches (and register indirect branches) are disallowed. Register indirect memory accesses have to be done by a routine within the OS that verifies that the access is legal. Essentially, register indirect memory accesses are interpreted [20].

We feel that the restrictions which would have to be placed on code so it can be inspected (in finite time!) are too limiting for the types of handlers we anticipate. For this reason, we will not consider code inspection any further.

A variation of this is proof carrying code [68]. The code to be inserted carries a proof that it does not violate the policies set forth by the executor (the kernel in our case). The proof should be easy to verify and guarantee that the code is safe to execute. This research is still in its very infancy, but might be applicable to small handlers.

Chapter 3

Related Work

Reference to and comments upon relevant work by others on the same or similar problems [55].

The high cost of protection domain crossings, especially from user-level into the kernel, has been well documented [65, 71, 2, 62]. Often, a mismatch between modern microprocessor architectures and the changing needs of newer operating systems is cited as the main reason. Hardware, as well as software solutions, have been proposed to alleviate the problem [22, 14, 43, 19, 54, 13, 56]. While some of the achieved improvements are significant, there is still a cost associated with a protection domain crossing. If that cost is high enough to cover the overhead of safely executing untrusted code in the kernel, then kernel extensions have merit. Otherwise, up-calls to the user level are more appropriate.

The research proposed builds on work done in several areas. In Section 3.1 we investigate several existing methods to send a message to a remote node and let a user-specified handler execute on the remote node when the message arrives. Essentially, all suffer from too high overhead or inflexibility. In Section 3.2 we look at operating systems that allow user processes to extend the kernel by inserting code. All of the extension methods described in Section 2.2 are used in one form or another. The goal of this research is to compare them and find the one that is best suited for our applications.

In Section 3.3 we look at two examples where hardware is used to solve the problem of high overhead in the start-up and execution of remote handlers. Since code is inserted into the kernel at runtime, research in the area of dynamic code generation is also of interest to us. We briefly discuss it in Section 3.4. Finally, we explore several areas related to interpreters in Section 3.5.

3.1 Remote Handlers

Remote handlers are used extensively in client-server applications and network file systems. With the advent of high-speed and low-latency networks it has become apparent that the software overhead required to transfer messages has to be reduced significantly to take advantage of the new hardware technologies. We look at active messages, Intel NX, and Puma portal event handlers in MP systems, as well as methods employed in workstation environments.

3.1.1 Active Messages

Active messages were introduced in [100] and have enjoyed considerable attention [98, 103, 80, 57, 89]. Especially on the CM-5, where the network interface can be mapped into the user's address space, active messages have a very small overhead compared to traditional message passing. This translates into very low message latencies. However, there are some characteristics that make active messages unsuitable as a general solution to the problem of fast responding handlers. For example, bandwidth is adversely affected by the necessity to always run a handler. A data transfer method that moves data without invoking a handler will get better bandwidth.

Most active message implementations use polling to achieve the low latencies reported in the literature. Adjusting the polling frequency introduces a tradeoff between fast response time and amount of overhead attributed to unneeded polling. In some cases polling complicates a program and in others it is not appropriate. For example, in a compute intensive application that spends much of its time in libraries such as the BLAS, polling can only occur between calls to the library; hence, handler response time is poor. Interrupt driven implementations of active messages suffer from context switch overhead.

Originally, active messages were designed for communication among the nodes of the same application. Sending the actual start address of a function combined with non-existant protection and recovery mechanisms, make active messages unsuitable for use between arbitrary applications, servers, and machines.

A new organization and application programming interface remedies the shortcomings of the original active message design [58], and tries to bring active messages into the mainstream. In principle, the receive part of an active message endpoint as described in [58] resembles a single block Puma portal with an attached handler. The handler is executed when the data has been deposited into the portal. A single block Puma portal consists of a memory descriptor specifying the start address and the length of a memory segment where data is to be stored or retrieved from.

This new model addresses security and flexibility concerns but only offers the traditional choices to handle incoming messages: polling or interrupt driven with the handler in the user's address space.

3.1.2 Intel NX

NX is the message passing system used on the Intel Paragon and its predecessors [73, 74]. The `hrecv()` function behaves much like an ordinary receive with the exception that, upon message arrival, a user specified handler is invoked. The mechanism is interrupt driven and a full context switch to the handler occurs on message reception.

To measure performance, we used a simple benchmark that posts an `hrecv()` and then waits for a global variable to change. The handler that is invoked by `hrecv()` increments this variable and returns. (This is a measurement of Δ_{resp}.) Averaged over 10000 trials, we measured an invocation time of about 300μs. The same measurement for an `irecv()`, a non-blocking receive without a handler, yields 14μs. Figure 3.1 shows the pseudo code of our benchmark.

In both cases the receive is pre-posted and the receiving node spins in a tight loop at user level, waiting for the incoming message. The incoming message causes an interrupt and the operating system kernel reads the zero length message from the network.

To process an `hrecv()`, the kernel switches context to the handler, executes it, and does another context switch to the original user program. The handler sets the global variable `gotit` to TRUE. Inside the loop, the test program checks the global variable to determine when the handler has run. While the test program spins inside the loop, it produces a series of time stamps. Once outside the loop, it is easy to determine the time stamp from just before the interrupt took place. The gap to the next time stamp is larger ($> \epsilon$) than the gaps measured in successive, uninterrupted loop iterations. The constant ϵ is larger than the overhead of a `clock()` function call, but less than the interrupt time: $(\texttt{clock()}_{t2} - \texttt{clock()}_{t1}) < \epsilon < (\Delta_{int} + \Delta_{reti})$

In the case of an `irecv()`, the kernel simply returns to the user level. Inside the loop, the test program uses `msgdone()` to know when the message has arrived. Again, a larger gap between time stamps reveals the one just before and the one just after the interrupt has been taken. The high cost of `hrecv()` makes it impractical to use in the contexts proposed for this research.

3.1.3 Puma Portal Event Handlers

Puma [105, 84] is an operating system specifically designed for high-performance MP architectures. On each node, the systems consists of a minimal kernel (the quintessential kernel), a process control thread (PCT) which is a trusted user-level process that establishes the policies on the node, and the libraries linked with each application.

Puma portals are openings into the applications's address space. Data structures, shared between the kernel and the user determine where incoming messages are to be deposited. The shared data structures consist of a portal table, matching lists, and memory descriptors. With the appropriate combination of these basic building blocks, a user-level process can describe the actions the kernel is to perform upon arrival of a message. Portals allow the construction of most higher-level message passing protocols in user space without costly memory-to-memory copies.

When a message arrives, the data and its header are placed into memory according to the application's specification in the portal structures. If a portal event

```
main()
{
    loop {
        if node 1 is ready then
            break
    }
    send zero length msg to node 1
}
```

(a) Node 0

```
handler()
{
    gotit = TRUE
}


main()
{
    gotit = FALSE
    hrecv(handler, ...)
    $t_1$ = $t_2$ = clock()
    loop {
        $t_1$ = clock()
        if gotit then
            if $t_2 + \epsilon > t_1$ then
                $t_2$ = clock()
            break
        $t_2$ = clock()
        if gotit then
            if $t_1 + \epsilon > t_2$ then
                $t_1$ = clock()
            break
    }

    if $t_1 > t_2$ then
        time = $t_1 - t_2$
    else
        time = $t_2 - t_1$
}
```

(b) Node 1

Figure 3.1: `hrecv()` **Benchmark:** Measure the time from message arrival until start of the handler (Δ_{resp}).

handler has been installed, then the kernel will transfer control to a user-level master handler. The master handler can be replaced by the application and is responsible to dispatch events to event handlers. This is similar to active messages, except that the data already resides in memory. Invoking the handlers requires a context switch from kernel to user level.

3.1.4 Workstation Environments

Workstation networks and interfaces are approaching the performance characteristics of MP systems [93]. However, network protocols have not kept up with this trend [1]. Illinois Fast Messages [72] and U-net [99] are two examples of approaches to reduce the software overhead.

Fast messages (FM) recognize the need of overlapping communication with computation, precise control of the hardware involved (busses and network interface), and efficient buffer management. Using Myrinet interfaces, this approach goes as far as replacing the program in the network interface coprocessor (LANai) with one that is specific to the FM protocol. Data is moved directly into user space, avoiding a costly memory-to-memory copy. The protocol further assumes a reliable network and puts the burden of message content verification (checksums) onto higher level layers. (Measurements under Amoeba indicate that user level protocols incur only small additional costs, but provide increased flexibility [70].)

The U-net approach is to map the network interface into user space and avoid protection domain crossings for message transfers. To send or receive a message, the kernel does not have to be invoked.

Both FM and U-net lower message passing latencies, but do not directly address the problem of handler response time. In both cases, a polling user application can quickly respond to user messages. A general, interrupt driven solution still has a high cost associated with it.

3.2 Extensible Operating Systems

An extensible operating system lets an application apply certain customization to tailor the operating system's behavior to the needs of the application. Applications running on personal computers have been taking advantage of the non-existing protection mechanisms in MS-DOS and the Apple Macintosh operating system to extend the operating system. For example, intercepting keystrokes and mouse events, as well as writing to screen memory directly is possible. These simple operating systems are not able to protect themselves or other applications from buggy or malicious user code.

Many variants of the Unix operating system allow a trusted user (the system administrator) to modify the running system by adding new device drivers and kernel services such as different file systems. Flexibility is lost, since the applications cannot dynamically adjust the system to their needs anymore. Furthermore, the operating system still has no way of verifying that the extension will not harm the system or other applications.

Current research in extensible operating systems tries to address these issues. The goal is to let applications safely modify system behavior. For example, it should be possible for an application to specify its own memory page manager. The kernel would call this page manager when the system needs to reclaim memory pages currently allocated to this application. The application specific page manager can then decide which pages should be given up. For this to be efficient, the page manager needs to reside inside the kernel, so no expensive cross-domain calls are necessary to evict memory pages. The page manager has to be isolated so it can not disrupt other kernel services or wreck havoc with the page handling of other applications.

Several recent systems use the methods presented in Section 2.2 to enable such extensions. In this section we look at SPIN, the MIT Exo kernel, GLUnix, VINO, and μChoices.

3.2.1 Spin

SPIN [7] is an extensible operating system that allows kernel extensions, so called spindles, to be inserted dynamically. Spindles as well as SPIN itself, are written in Modula-3, a type-safe object oriented programming language. The use of a type- and pointer-safe language prevents spindles from calling services inside the kernel that have not been specifically exported. The language makes it also impossible to access memory that is not part of an object to which the spindle has been given explicit access.

As long as the Modula-3 compiler is trusted to implement the language specification faithfully, and only spindles generated by this compiler are accepted, SPIN is safe from malicious code. The compiler runs at user level and is the only process that is allowed to insert spindles into the kernel. Only spindles generated at runtime can be inserted into the kernel. This eliminates the need to cryptographically sign a spindle, but has the drawback that the time to compile and optimize a spindle has to be expended for each spindle insertion. Dynamically linking a spindle into the running kernel also takes time. It is assumed that the time savings and flexibility of having the spindles execute inside the kernel compensate for this overhead.

3.2.2 Exo Kernel

The MIT Exo kernel [28, 27, 29] is an extreme approach to operating systems design. It attempts to lower the operating system interface to the hardware level, eliminating all abstractions that traditional operating systems provide, and concentrating on multiplexing the available physical resources.

All work traditionally done inside the kernel to provide abstractions, such as memory mapped I/O and complex thread packages, is moved into application-level software layers. The kernel simply allocates, deallocates, and multiplexes physical resources, for example memory, time-slices, access to I/O devices, disk storage, etc. This is similar to the Puma kernel approach, where many of the abstractions are pushed into the PCT (Process Control Thread) or user-level libraries. The Exo kernel takes this to an extreme, since all abstractions are removed from the kernel and no privileged user-level processes, such as the PCT, are allowed; every abstraction is provided by the application (usually in the form of a library). This allows applications to customize abstractions, choose the best fitting one among several, or circumvent libraries that are not efficient enough for the task at hand.

In principle, there should be fewer traps into the kernel, since most of the OS functionality is at the user-level. The traps should also be cheaper, since there are fewer services to dispatch inside the kernel. For cases such as TLB miss handling, user-level code can be inserted into the kernel. A combination of code inspection and sandboxing is used to insert untrusted user code safely into the kernel.

3.2.3 GLUnix

The Global Layer Unix (GLUnix [97]) system uses software-based fault isolation (SFI) to move operating system functionality into user level libraries. User applications are linked with the operating system libraries, and system calls are converted to function calls into these libraries. Proponents of GLUnix claim that the cost of SFI is offset by not having to trap into the kernel for every system call.

The GLUnix libraries will provide a single system view of a network of workstations, even though the individual workstations run a standard operating system. This approach is limited by the functionality exported by the underlying operating system. Furthermore, to send and receive messages, the services of the operating system kernel are required. Since the goal of GLUnix is to use vendor supplied operating systems and use them unchanged by building additional functionality in a layer that sits on top of the operating system, there is no possibility of inserting user handlers into the kernel, unless the underlying operating system supports that.

3.2.4 VINO

The VINO kernel [85, 82, 83] is designed as a platform for database management systems. From the outset, VINO is designed to let applications specify the policies the kernel uses to manage resources. A further goal is to make kernel primitives accessible to the user level. For example, synchronization functions the kernel uses, might be useful to applications as well.

Applications establish a resource management policy by inserting a *graft* into the VINO kernel. Grafts are written in C or C++. The compiler inserts range check instructions for all memory accesses, similar to segment matching in Wahbe et al. [102], and ensures that no privileged instructions, such as the disabling of interrupts, are issued. The generated code is then marked with an encrypted fingerprint (signature) that is verified by the kernel during code insertion.

Since grafts are used to implement policies, access to kernel functions and data, for example locks, is a must. To prevent a graft from holding a lock indefinitely, transaction techniques are used. If a graft has to be aborted because of an error or because it ran too long, its actions can be undone and locks held by the graft can be released.

As in SPIN, some trust is placed into the compiler. To avoid a type-safe language, software based fault isolation is used to protect against illegal memory accesses.

3.2.5 μChoices

In the μChoices operating system [11, 92] so called agents can be inserted into the kernel. Agents are written in a simple, flexible scripting language similar to TCL, and are interpreted. Agents batch a series of system calls into a single procedure that requires only one trap into the kernel to be executed. Agents use existing kernel services and do not extend the functionality of the kernel or provide services that are not available at user level. Agents are a simple optimizations to eliminate the overhead of several system calls.

Methods to safely execute untrusted code in a privileged environment are compared in [86]. Among the methods chosen is interpreted TCL because of its mention in the μChoices papers. As might be expected, TCL's execution speed is orders of magnitudes slower than some of the other methods. Therefore, the authors claim that interpretation in general is too slow. However, TCL interpreters are not optimized for speed. Furthermore, with appropriate restrictions to the language and application of advanced interpretation techniques, it might be possible to speed-up interpretation considerably.

3.3 Hardware Solutions

There are many projects that use hardware to make context switches and message reception faster. The Stanford FLASH multiprocessor and at the MIT J-machine aim to provide fast hardware to accommodate many different message passing paradigms and use software handlers to act on incoming messages.

3.3.1 Stanford Flash

In the Stanford FLASH multiprocessor architecture [40, 53, 39] each node contains a custom node controller called MAGIC. MAGIC is located between the network and the CPU and memory. It consists of several queues and a protocol processor. The protocol processor has instruction and data caches, independent of the main processor on the board.

The protocol processor uses physical addresses to access main memory. Translation of virtual addresses and the necessary verifications are performed when addresses are transferred from the main processor to the protocol processor inside

MAGIC. This gives the protocol processor the ability to transfer data to and from main memory very efficiently without affecting the operation of the main processor. That is, no interrupts or context switches occur on the main processor when new messages arrive. The design is very flexible and allows the implementation of a wide variety of protocols inside MAGIC.

MAGIC handlers directly control the message passing behavior of a node. Erroneous handlers executing on the protocol processors can corrupt data and crash the whole machine. Therefore, handlers have to be created and validated with the same scrutiny as the hardware in the system, making it impossible to run user-created handlers on the protocol processor. It might be possible to run the R-code interpreter on the protocol processor. This would allow users to submit code that runs (interpreted) on the protocol processor.

3.3.2 J-Machine

The MIT J-machine [17, 69] combines a general purpose CPU with a network controller. (Similar to the nCUBE processors [67, 24].) Upon arrival of a message on a J-machine node, the CPU dispatches a handler to process the message. The handlers are fine-grained threads. Dispatching is done by the hardware and, therefore, extremely fast (less than $1\mu s$).

In contrast to the Stanford FLASH project, there is only one CPU to handle message traffic and user applications. On the other hand, user applications and message processing are tightly integrated, and applications provide the threads to handle incoming messages. From an applications point of view, this makes the J-machine more flexible, since the protocol processor code in the MAGIC cannot be changed by an ordinary user.

3.4 Dynamic Code Generation

Usually, executable code is generated by a compiler before the executable is loaded and run. A technique called dynamic or runtime code generation, delays compilation until the executable is already running. For example, a function to perform a matrix multiply is not compiled into its final form until the sizes of the two matrices to be multiplied are known. The sizes can be expressed as constants in the arithmetic routine and the compiler can perform certain optimizations that would not be possible if the sizes were unknown and had to be expressed as variables. Even with the overhead of the compilation step at runtime, it is sometimes worthwhile to consider dynamic code generation, especially in the case when a function will be executed often, after it has been compiled once [46, 44, 45, 47, 60, 30, 26, 41, 59, 25].

Inserting code at runtime into the kernel is a form of dynamic code generation. The application can manipulate the R-code image, or even compile a higher-level language into R-code, before it inserts the R-code image into the kernel. For example, a broadcast function could hard-code the destination nodes into the R-code image before it is inserted into the kernel. The image would be slightly different on each node, but would not have to do any computations to determine the destinations of a broadcast.

3.5 Interpreters

Because of their simplicity, interpreters are often employed early on as a proof of concept, to be later replaced by compiled languages. Nevertheless, some applications, such as the BSD packet filters (Subsection 3.5.4) and Java applets (Subsection 3.5.5) over the world wide web (WWW), exploit interpreter characteristics in areas where speed matters.

First, we look at techniques that have been developed to speed-up interpretation (Subsection 3.5.1). Since most interpreters use a (virtual) stack machine to execute the programs, we look at work done on implementing stack machines in Subsection 3.5.2. Many of the interpretation techniques made their first appearance in Forth systems. We look at Forth in Subsection 3.5.3. In Subsection 3.5.4 and 3.5.5 we look at uses of interpreters in the BSD packet filter and in Java.

3.5.1 Interpretation Techniques

Interpretation has several desirable characteristics. Interactive systems benefit from a quick turn-around time and the extensibility of the interpreters themselves. For example, interpretation of commands can begin as soon as the user starts typing. There is no edit, compile, test cycle. Some interpreters can be extended by the user. Forth, for example, allows the definition of new keywords by the user. These new keywords are integrated into the running interpreter and can be used like any other keyword predefined by Forth.

Interpreters are relatively easy to write (when compared to a compiler) and are easy to port. Furthermore, intermediate code representation, bytecode for example, makes "precompiled" executables very small. Often much smaller than the corresponding source and even a compiled binary.

The only real drawback usually is execution speed. Simple interpreters are often ten to hundred times slower than the same program written in a language such as C or Pascal and executed as a native binary. Several techniques exist to make interpretation fast. One of the earliest techniques is Bell's threaded code technique [6]. The idea is to preprocess the code and convert language statements into subroutine calls. In principle, this reduces the execution overhead of an interpreter to one or two assembly instructions per interpreted statement.

Indirect threaded code [21] adds a level of indirection and makes the technique more portable. Kogge [49] reviews threaded code techniques and compares them. He finds that performance penalties for threaded code versus direct assembly coding are about 1.2:1 for minicomputers. A paper by Paul Klint [48] compares interpretation techniques. He finds that instruction fetch time of direct threaded code is faster than indirect threaded code. Both threaded code techniques are much faster than a traditional interpreter that uses opcode tables. As the complexity of the interpreted instructions increases; i.e. the subroutines that implement the instruction become larger and more complex, the less relevant the instruction fetch time becomes.

Cint [18], a C language interpreter, shuns threaded code techniques, claiming they are machine dependent. Instead, Cint relies on a minimal (RISC) virtual machine for speed. The instruction set consists of 49 executable instructions and 14 pseudo-operations. The overhead of executing a C program under Cint compared to the execution of a compiled binary, is considerable: On average 29 times slower on a VAX-11/780 and 36 times slower on a Sun-3/75. To avoid the machine dependency problem, Ertl [31] uses a feature of GNU gcc [90] to generate a threaded code interpreter from a C language source. GNU gcc extends the C language and allows labels to be treated as values. Together with GNU gcc's computed goto statement it is possible to write a threaded code interpreter without resorting to machine language. The interpreter is portable to any system that supports GNU gcc.

For our research we will use threaded code techniques. If possible, we use the C language extensions of GNU gcc to remain as portable as possible. At the same time we will adapt and modify R-code so that an interpreter for it can be small and simple (like Cint), and keep the amount of computation done per interpreted instruction high, so the instruction fetch time can be amortized [78].

3.5.2 Stack Machines

It is easy to map an arithmetic expression onto a stack-based virtual machine and then evaluate it. Many interpreters, specifically Forth, are built on stack-based virtual machines. To speed up execution, some stack-based designs have been implemented in hardware [51]. Microprocessors have been built for the direct execution of Forth [38, 64] and are being designed for Java [52]. A large body of theoretical work on how to efficiently execute stack-based programs on register-based CPUs [37, 32, 34, 33, 75] and on optimizing code for stack machines [50, 15, 10] exists. The R-code interpreter will be stack-based and the techniques for optimizing stack-based programs and to speed up stack-based interpreters will be applied.

3.5.3 Forth

Forth [79] is a stack-oriented language. It was first used to control telescopes and process astronomical data. Since then, it has found a wide spectrum of uses, but is still often used in embedded systems. For example, the boot monitor of Sun workstations uses Forth to interact with the system administrator.

Forth interpreters are very fast. One reason for this is that Forth is relatively low-level and gives the programmer many opportunities to optimize stack operations. Other reasons are the innovative interpretation techniques that first appeared in Forth. The first Forth system employed indirect threading as later described in [21].

Forth allows a user to define new words that are dynamically integrated into the interpreter. New words can be accessed and executed as fast as the builtin words. Therefore, it is very easy and efficient to customize the interpreter to specific tasks at hand.

We considered to make Forth the language that is used to interpret code inside the kernel. We decided against it, because many features of Forth are geared at interactive use and I/O. Stripping these from Forth would leave a language very similar to R-code. Since we want to optimize R-code for fastest possible interpretation and a well suited target language for a C compiler, we decided to start from scratch. R-code has to evolve and, in its final form, may not look like Forth at all, even if we had started there.

3.5.4 BSD Packet Filter

The BSD packet filter originated in the Xerox Alto. The motivation, implementation, and performance on various BSD machines is described in [66]. Packet filters are written in a simple stack-based language and inserted into the running kernel. For each incoming data packet, the filters are executed until one accepts the packet. There are no branch instructions and only a rudimentary set of operations. Words at constant offsets in the packet can be examined using comparisons and the three boolean operators `AND, OR`, and `XOR`. No other arithmetic operations are available.

Interpretation was chosen to make it possible to move packet filters from user space, with the associated context switches and kernel traps, into the kernel, saving that overhead. This enhanced performance considerably. The interpreter ensures that only words inside a packet are accessed and aborts faulty handlers (after a stack underflow for example). In general, though, security is not enforced. Any filter can accept any packet.

An improved version of the packet filter is presented in [61]. The instruction set has been extended and now includes arithmetic operators such as `add, sub, mul`, and `div`, as well as conditional branch instructions. The pseudo machine is now register based. It consists of an accumulator, an index register, a scratch memory store, and an implicit program counter. The authors claim that such a machine can be simulated faster on today's register-based RISC architectures. The performance gain over the earlier implementation seems to validate this claim. However, it may be that the language used in the first BSD implementation is just too simple. For example, it is not possible to read a word from the packet, mask certain bits, and then use this intermediate result several times in comparisons. Instead, the first BSD implementation needs to read the same word every time and has to redo any masking done before. The newer implementation can easily store the intermediate result and reuse it when necessary. Therefore, the claim that a stack-based virtual machine is not as suitable as a register-based virtual machine for modern CPUs is not substantiated.

Performance of both BSD packet filters degrades as more filters are added, since each is executed sequentially and independently. With several sessions active at the same time, many filters that differ only minimally (in matching the destination port number for example), have to be installed. This problem is addressed in [107]. The new filter mechanism is called MPF (Mach Packet Filter) and is an extension of the register-based packet filter [61].

To achieve scalability in the presence of many packet filters, MPF offers a new instruction that is reminiscent of a C `switch` statement. It replaces an instruction sequence in the register-based packet filter that is used to dispatch among various protocols. MPF collapses the new instruction present in all filters into a single, fast, dispatch routine. Therefore, no matter how many filters are present, the common dispatch code is executed only once. Another reason this is faster, is that several instructions can be replaced by a single virtual machine instruction. This lowers the overhead of virtual machine instruction dispatch.

3.5.5 Java

Java [63, 36] is an interpreted, object-oriented language with a syntax similar to C and C++. Recently it has attracted much attention through its use in WWW browsers. Applets, written in Java and translated into bytecodes, are made available on WWW servers. A client, the WWW browser, fetches the applet and executes it on the client side. Since the execution is local, applets can create graphical effects, for example, without consuming large amounts of bandwidth to the server.

Applets can be written by anybody and made available anywhere in the world, and cannot be trusted. The browser on the client side interprets the applet and monitors disk accesses, network communication, and other activities of the applet. An interpreted language is ideal for this purpose, since any desired restrictions can be enforced by the interpreter.

Transferring untrusted applets into an environment in which execution could cause trouble is similar to the idea of executing user code inside an operating system kernel. Actual transfer from one node to another in an MP system is not necessary, since all nodes with the same application have copies of the same executable. Also, the kernel knows the user process's owner. (This does not mean user code can be trusted.) Java is a general-purpose high-level language. It offers many features that are not necessary for kernel embedded handlers. While R-code should be general-purpose, it does not need to be able to deal with an interactive user and do file I/O, for example.

Chapter 4

Proposed Solution

A statement or characterization of what kind of solution is being sought [55].

Among the four possible ways, outlined in Section 2.2, to safely introduce untrusted user level code into the kernel, we believe that a kernel embedded interpreter is the most efficient and best suited approach for event handlers. Therefore, the dissertation will attempt to prove the following thesis:

A kernel embedded interpreter is an effective way to decrease the latency of user-level communication primitives.

It should be possible to write the handlers in C (or any other high-level language). The handler is then compiled into what we call R-code[1]. During code insertion, the kernel transforms R-code into direct threaded code. To avoid a parser and assembler in the kernel, we transform R-code into a binary representation before it is inserted into the kernel. The assembly of R-code into its binary representation can be done by the application prior to insertion into the kernel, or off-line when the application is compiled. The former has the advantage that the application can easily make runtime changes, for example to insert constants, such as the logical node number. It has the disadvantage that the R-code assembler has to be resident on the node. The latter method avoids the overhead of last minute assembly, but makes it harder to make modifications at runtime. The types of code and the transformations are shown in Figure 4.1.

R-code has to have several characteristics. Most importantly, it has to be possible to interpret it at speeds that are close to assembly code generated by standard C compilers. R-code and the interpreter for it have to be able to take advantage of certain Puma kernel characteristics, as well as exist within the limitations imposed by the kernel (we will discuss them in the next paragraph). Another objective for the design of R-code is that it should be relatively simple for a C compiler to

[1]The name R-code was chosen since P-code [106] and U-code [12] already exist. Any relationship to the author's initials is purely coincidental.

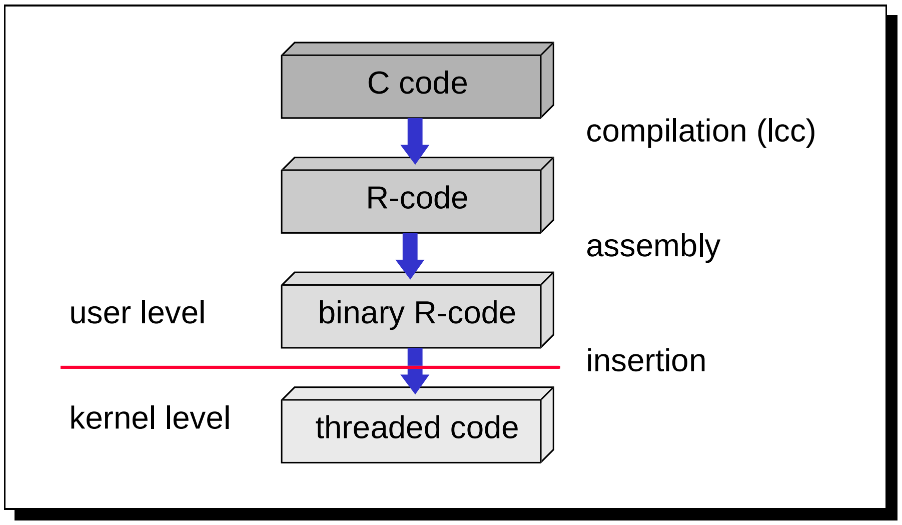

Figure 4.1: **Code Transformations**: From C code to threaded code.

produce efficient R-code for the handlers we discussed in Chapter 1. We expect R-code to evolve during this research, swinging between easy to compile to and fast to interpret.

The Puma kernel has a few unique characteristics that have to be taken into consideration when designing R-code. There is no demand paging under Puma. Therefore, all memory pages are always present. This gets around the problem of having to bring in a page for a message that is currently being received or sent. It means that the Puma kernel can always stream messages directly into user memory, with no need for buffering and costly memory copies. Puma portals are currently added to the Linux kernel. The assumption that the necessary pages are always in physical memory is no longer true in that environment. Our first implementation requires that memory which is overlayed by a portal, must be wired down. It has to be made unpagable before the Puma portal can be activated.

For the R-code interpreter this implies that it can be very simple and the individual handlers can be executed to completion on each message arrival. No blocking has to occur because of missing virtual memory pages. There will be a time limit for each handler invocation to prevent a handler from consuming all CPU cycles of a node in an infinite loop. While the Puma kernel is executing, all interrupts are disabled. In Puma it is therefore difficult to enforce a running time limit on code that is executing inside the kernel. For interpreted code it is easy however. The interpreter simply counts the number of instructions executed and aborts after a predetermined limit is reached.

The Puma kernel and its data structures are static in size. This means that the room to insert user handlers is of fixed size. The stacks of the virtual machine as well as the data segment of the handler are of fixed size. This should not pose any problems. The handlers and their local data will be small. Large handlers will also run longer and belong into user space.

And last but not least, the Puma kernel is non-blocking. There are no functions in the kernel that have to be suspended because a resource is busy. This means that handlers have to be given a certain (small) amount of time to run and then be aborted if that limit is exceeded. Most handlers will probably never reach that limit. Those that do, might be better implemented as up-calls to a user-level handler. We will determine the cross-over point, where it becomes more efficient to do an up-call rather than using a kernel extension, when we do our experiments.

Handlers should have access to the header of the message that triggered invocation of the handler, as well as the data delivered by the message. Aside from the data stack, there should be a fixed-size data area for each handler that allows storage of data between invocations of the same handler. Global sum operations are an example where this storage becomes important. On the first invocation the handler stores the value. On the second invocation it sums it with the newly arrived value and sends the sum to the parent in the fan-in tree.

Handlers should also have access to the memory owned by the process that installed the handler. This allows handlers to manipulate counters and set flags inside the application. In the C code for the handler these memory references should be marked as external. A good way of linking these references to actual memory locations in the running application has to be found.

Handlers that fault (on division by zero for example) or attempt unsafe operations (illegal instructions or illegal memory accesses) are terminated. A return code associated with the handler should indicate the error. For debugging purposes and performance tuning as much information as possible about the handler run should be made available to the application that inserted the handler. This could be achieved by having two interpreters in the kernel: A high-performance one, and another that logs information such as running time, register usage, external memory accesses, etc.

The interpreter has to be very fast. We currently plan a stack-based virtual machine which introduces the problem of mapping a stack onto a register file. There has been some research in this area and we intend to find a good solution for the R-code interpreter. If necessary, we may consider to change to a register-based virtual machine. At the present time this seems unlikely, though.

Chapter 5

Statement of Work

A plan of action for the remainder of the research [55].

To validate that an interpreter is fast enough to support handlers inside the kernel, we will compare a kernel embedded interpreter approach with a trusted compiler solution as well as software based fault isolation. The measurements will be performed using the Puma operating system. This system is currently running on the Intel Paragon using Intel i860 RISC CPUs. It also runs on the Intel Teraflop architecture which uses Intel Pentium Pro (P6) CISC CPUs.

Puma will be modified to let a user program install a portal event handler at runtime. The handler can be written in R-code and will be interpreted when a message arrives. The handler can also be a binary that has undergone software based fault isolation. Alternatively, the handler can simply be trusted, assuming it came from a trusted compiler. In the latter two cases no interpretation during execution time takes place; the inserted handler is executed directly. Baseline comparisons will be made against the standard user-level portal event handlers of Puma. A handler that takes longer to execute inside the kernel, than the same handler at user level including the context switch overhead, should probably be run at user level.

During the insertion of software isolated code, the algorithm described in [102] should be applied. However, since we are mainly interested in the measurement of execution time, not code insertion time, we will most likely forgo this verification step.

The simplest way to integrate code from a trusted compiler, is to directly link it into the kernel. Insertion then consists of selecting the appropriate routine to be executed on message arrival. This is a sufficient procedure to measure execution time of such handlers. It does not address the cost of linking and binding, nor the runtime system overhead of a type-safe language. Linking and binding costs are paid at code insertion and will be more or less ignored by this research. Runtime system overhead has to be taken into consideration, though. All handlers, whether trusted, interpreted, or software fault isolated, should be represented by the same C source to make the comparison fair. To estimate the runtime system overhead of a type-safe language, we will modify the C compiler to insert instructions that model bounds checks.

We will measure the performance of representative handlers for broadcast, MPI, Split-C, and Cilk. Ideal test conditions require that the same source code is used for each handler in any of the three insertion methods. For this reason we will use the retargetable C compiler *lcc* [35], and modify it in the following manner.

For the first experiment, we will retarget lcc so that it produces R-code. R-code will be the same for the i860 and x86 architecture. There exists already a back-end for lcc that produces x86 code. A member of the Puma team is currently building a back-end for the i860. For the second experiment, these back-ends will be modified to insert software isolation code before each load, store, and jump instruction. This will produce software isolated code. For the third set of experiments we will modify the x86 and i860 back-ends to introduce range checks, and other runtime checks typically performed by the runtime environment of a type-safe language such as Modula-3 used in the SPIN project.

Therefore, the same compiler (with different back-ends) will be used to generate code for all three techniques on a RISC and a CISC architecture. The operating system and the test applications are going to be the same for all six possible configurations. Table 5.1 shows all six configurations under which each test will be performed.

Table 5.1: Test Configurations

	Paragon, i860	Tflops, x86
interpreted	$\surd$	$\surd$
SFI	$\surd$	$\surd$
trusted	$\surd$	$\surd$
up-call	$\surd$	$\surd$

To determine the tradeoff between kernel insertion and up-calls, we will also measure the cost of using the Puma portal event handler up-call mechanism. Again, the handlers will be compiled using lcc. No protection mechanisms will be compiled into these handlers, since they run at user level.

Table 5.2 lists the steps to be performed and approximate completion dates:

Table 5.2: Project Milestones

Completion Date	Task
4/1/96	Initial design of R-code completed plus set of tools (non-optimized interpreter, assembler, disassembler).
5/1/96	Retarget lcc to produce i860 code.
5/1/96	Retarget lcc to produce R-code.
9/8/96	Puma kernel able to insert R-code and interpret it on message arrival.
10/1/96	Optimized i860 versions of kernel interpreter.
10/13/96	R-code modified to be best suitable as a target for lcc.
10/17/96	i860 back-end of lcc modified to insert software isolating instructions.
11/1/96	i860 back-end modified to generate runtime checks for type-safe languages.
11/1/96	Execution of software isolated code in i860 kernel enabled.
11/15/96	x86 back-end of lcc modified to insert software isolating instructions.
11/15/96	Optimized x86 versions of kernel interpreter.
11/22/96	x86 back-end modified to generate runtime checks for type-safe languages.
12/1/96	Execution of software isolated code in x86 kernel enabled.
12/15/96	Test suite of handlers selected and compiled for all targets.
12/22/96	Up-call mechanism in i860 kernel enabled.
1/9/96	Up-call mechanism in x86 kernel enabled.
2/1/96	Performance measurements completed on all targets.
3/1/96	Dissertation written.

Chapter 6

Acknowlegements

Most of all I need to thank my advisor A. Barney Maccabe for suggesting the persuasion of a Ph.D. (and then actually accepting me as his graduate student). Next, I like to thank the members of my dissertation committee for their acceptance of this task and their helpful comments and suggestions in preparing this proposal: Charlie Crowley, Charles E. Leiserson, and Bernard M. E. Moret.

Special thanks go to Ron Brightwell, David Greenberg, and Kevin Mccurley for their careful reading of the proposal manuscript and their suggestions for improvement. Mack Stallcup wrote a lcc backend for the i860. That is much appreciated. Last, but not least, I like to thank all the Puma team members at Sandia National Laboratories and the University of New Mexico for their support. Specifically, I thank David van Dresser, Lee Ann Fisk, Tramm Hudson, Chu Jong, Michael Levenhagen, and Lance Shuler for letting me discuss the ideas in this proposal with them, and acknowledge their input, help, and setting me straight where necessary.

Bibliography

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In ASPLOS 91 [4], pages 108–120.

[3] *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Palo Alto, CA, October 1987. ACM Press, New York. Published as SIGPLAN Notices, volume 22, number 10.

[4] *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, CA, April 1991. ACM Press, New York. Published as SIGPLAN Notices, volume 26, number 4.

[5] *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 1994. ACM Press, New York. Published as Operating Systems Review, volume 28, number 5.

[6] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.

[7] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN - an extensible microkernel for application-specific operating system services. Technical Report CSE-94-03-03, University of Washington, February 1994.

[8] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP 15 [88], pages 267–284. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In PPOPP 5 [77], pages 207–216. Published as ACM SIGPLAN Notices, volume 30, number 8.

[10] J. L. Bruno and T. Lassagne. The generation of optimal code for stack machines. *JACM*, 22(3):382–396, July 1975.

[11] Roy H. Campbell and See-Mong Tan. μChoices: An object-oriented multimedia operating system. In HOTOS 5 [42], pages 90–94.

[12] Paul Chan, Manoj Dadoo, and Vatsa Santhanam. Evolution of the U-code compiler intermediate language at Hewlett-Packard. In USENIX Summer 1990 [94], pages 199–210.

[13] David R. Cheriton. An experiment using registers for fast message-based interprocess communication. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 12–20, Brenton Woods, New Hampshire, October 1983. Published as ACM Operating Systems Review, SIGOPS, volume 17, number 5.

[14] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the tenth ACM Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, Washington, December 1985. Published as ACM Operating Systems Review, SIGOPS, volume 19, number 5.

[15] John Couch and Terry Hamm. Semantic structures for efficient code generation on a stack machine. *IEEE Computer*, 10(5):42–48, May 1977.

[16] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[17] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Greogory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, pages 23–38, April 1992.

[18] Jack W. Davidson and Richard A. Vaughan. The effect of instruction set complexity on program size and memory performance. In ASPLOS 87 [3], pages 60–64. Published as SIGPLAN Notices, volume 22, number 10.

[19] Jack W. Davidson and David B. Whalley. Methods for saving and restoring register values across function calls. *Software, Practice and Experience*, 21(2):149–165, February 1991.

[20] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *IFIP Congress 71*, pages TA–3–7 – TA–3–12, Ljubljana, Yougoslavia, August 1971.

[21] Robert B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[22] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In SOSP 14 [87], pages 189–202. Published as ACM Operating Systems Review, SIGOPS, volume 27 number 5.

[23] Charles B. Duff. Designing an efficient language. *Byte*, 11(8):211–224, August 1986.

[24] Bob Duzett and Ron Buck. An overview of the nCUBE 3 supercomputer. In H. J. Siegel, editor, *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 458–464, McLean, Virginia, October 1992.

[25] Dawson R. Engler. VCODE: A retargetable, extensible. very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 160–170, Philadelphia, PA, May 1996.

[26] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for efficient, machine-independent dynamic code generation. Technical Report ???, MIT LCS, 1995.

[27] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of HotOS V*, Orcas Island, WA, May 1995.

[28] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In SOSP 15 [88], pages 251–266. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[29] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole, Jr. The operating system kernel as a secure programmable machine. *OSR Special Issue on Extensible Operating Systems*, 1995.

[30] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generaton system. In ASPLOS 94 [5], pages 263–272. Published as Operating Systems Review, volume 28, number 5.

[31] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, 1993.

[32] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 315–327, 1995.

[33] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technische Universität Wien, Austria, 1996.

[34] M. Anton Ertl and Martin Maierhofer. Translating forth to efficient C. In *EuroForth '95 Conference Proceedings*, Schloss Dagstuhl, Germany, 1995.

[35] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.

[36] James Gosling and Henry McGilton. The Java language environment. http://java.sun.com/doc/language_environment/, October 1995.

[37] Makoto Hasegawa and Yoshiharu Shigei. High-speed top-of-stack scheme for VLSI processor: A management algorithm and its analysis. In *The 12th Annual International Symposium on Computer Architecture (ISCA)*, pages 48–54, Boston, Massachusetts, June 1985. IEEE Computer Society Press. Published as SIGARCH Newsletter, volume 13, issue 3.

[38] John R. Hayes, Martin E. Fraeman, and Robert L. Williams Thomas Zaremba. An architecture for the direct execution of the Forth programming language. In ASPLOS 87 [3], pages 42–49. Published as SIGPLAN Notices, volume 22, number 10.

[39] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the stanford FLASH multiprocessor. In ASPLOS 94 [5], pages 38–50. Published as Operating Systems Review, volume 28, number 5.

[40] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Henessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In ASPLOS 94 [5], pages 274–285. Published as Operating Systems Review, volume 28, number 5.

[41] Urs Hölze and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, FL, June 1994. Published as SIGPLAN Notices, volume 29, number 6.

[42] *Fifth Workshop on Hot Topics in Operating Systems (HotOS V)*, Orcas Island, WA, May 1995. IEEE Computer Society.

[43] Paul A. Karger. Using registers to optimize cross-domain call performance. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 194–204, Boston, MA, April 1989. ACM Press, New York. Published as SIGPLAN Notices, volume 24, number 5.

[44] Raghu R. Karinthi and Mark Weiser. Incremental re-execution of programs. In Symposium on Interpreters and Interpretive Techniques [91], pages 38–43.

[45] David Keppel. A portable interface for on-the-fly instruction space modification. In ASPLOS 91 [4], pages 86–95. Published as SIGPLAN Notices, volume 26, number 4.

[46] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report CSE-91-11-04, University of Washington, 1991.

[47] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report CSE-93-11-02, University of Washington, 1993.

[48] Paul Klint. Interpretation techniques. *Software Practice and Experience*, 11:963–973, 1979.

[49] Peter M. Kogge. An architectural trail to threaded-code systems. *IEEE Computer*, pages 22–32, March 1982.

[50] Philip John Koopman, Jr. A preliminary exploration of optimized stack code generation. *Journal of Forth Applications and Research*, 6(3):241–251, 1994.

[51] Phillip Koopman. *Stack Computers: The New Wave.* Ellis Horwood, 1989.

[52] Douglas Kramer. The Java platform: A white paper. http://java.sun.com:80 /doc/whitePaper.Platform, May 1996.

[53] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennesy. The stanford FLASH multiprocessor. ???, 1994.

[54] Butler W. Lampson. Fast procedure calls. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 66–76, Palo Alto, CA, March 1982. ACM Press, New York. Published as SIGPLAN Notices, volume 17, number 4.

[55] H. C. Lauer. On Ph.D. thesis proposals in computing science. *The Computer journal.*, 18(3), 1975.

[56] Jochen Liedtke. On μ-kernel construction. In SOSP 15 [88], pages 237–250. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[57] Lok Tin Liu and David E. Culler. Evaluation of the Intel Paragon on active message communication. In *Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995. Available from: http://www.cs.berkeley.edu/ ltliu/papers/isug95/isug_1.html .

[58] Alan Mainwaring and David Culler. Active messages: Organization and applications programming interface. version 2.0. Available from http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps, November 1994.

[59] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[60] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 191–201, Litchfield Park, Arizona, December 1989. Published as ACM Operating Systems Review, SIGOPS, volume 23, number 5.

[61] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, San Diego, CA, Winter 1993. USENIX.

[62] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In USENIX Winter 1996 [95].

[63] Sun Microsystems. The Java langauge: An overview. http://java.sun.com: /doc/white_papers.html, 1995.

[64] Daniel L. Miller. Stack machines and compiler design. *Byte*, 12(4):177–185, April 1987.

[65] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In ASPLOS 91 [4], pages 75–84.

[66] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, Texas, 1987. Published as ACM Operating Systems Review, SIGOPS, volume 21, number 5.

[67] nCUBE, 1825 NW 167th Place, Beaverton, OR 97006. *nCUBE 2 Processor Manual*, December 1990. PN 101636.

[68] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *The Secomd Symposium on Operating Systems Design and Implementation (OSDI'96) Proceedings*, pages 229–243, Seattle, Washington, October 1996.

[69] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 224–35, San Diego, California, May 1993. ACM Press, New York. Published as ACM Computer Architecture News, SIGARCH, volume 21, number 2.

[70] Marco Oey, Koen Langendoen, and Henry E. Bal. Comparing kernel-space and user-space communication protocols on Amoeba. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 238–245, Vancouver, Canada, May 1995. IEEE, IEEE Computer Society Press.

[71] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In USENIX Summer 1990 [94], pages 247–256.

[72] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Supercomputing '95: Proceedings*, 1995.

[73] Paul Pierce. The NX message passing interface. *Parallel Computing*, 1993.

[74] Paul Pierce and Greg Regnier. The Paragon implementation of the NX message passing interface. In *SHPCC 94*, 1994.

[75] Christian Pirker. *Üebersetzung von Forth in Maschinensprache.* Masters thesis, Technische Universität Wien, Austria, 1996.

[76] Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In Symposium on Interpreters and Interpretive Techniques [91], pages 150–152.

[77] *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, Santa Barbara, CA, July 1995. Published as ACM SIGPLAN Notices, volume 30, number 8.

[78] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, California, January 1995. ACM Press.

[79] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The evolution of Forth. *ACM SIGPLAN Notices*, 28(3):177–199, March 1993.

[80] Rolf Riesen, Arthur B. Maccabe, and Stephen R. Wheat. Active messages versus explicit message passing under SUNMOS. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference*, pages 297–303, June 1994. Available from: file://www.cs.sandia.gov /pub/sunmos/papers/ISUG94-2.ps.Z .

[81] Dennis M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, March 1993.

[82] Margo Seltzer, Christopher Small, and Keith Smith. The case for extensible operating systems. Technical Report TR-16-94, Harvard University, 1995.

[83] Margo Seltzer, Christopher Small, and Michael D. Smith. Symbiotic systems software. In *First Annual Workshop on Compiler Support for System Software*, February 1996.

[84] Lance Shuler, Rolf Riesen, Chu Jong, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma operating system for massively parallel computers. In *Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995. Available from: file://www.cs.sandia.gov/pub/sunmos/papers/puma_isug95.ps.Z .

[85] Christopher Small and Margo Seltzer. VINO: An integrated platform for operating system and database research. Technical Report TR-30-94, Harvard University, 1994.

[86] Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In USENIX Winter 1996 [95].

[87] *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993. Published as ACM Operating Systems Review, SIGOPS, volume 27 number 5.

[88] *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[89] Ellen Spertus and William J. Dally. Evaluating the locality benefits of acive messages. In PPOPP 5 [77], pages 189–198. Published as ACM SIGPLAN Notices, volume 30, number 8.

[90] Richard M. Stallman. *Using and Porting GNU CC*, November 1995. Version 2.7.2.

[91] *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, June 1987.

[92] See-Mong Tan, David K. Raila, and Roy H. Campbell. An object-oriented nano-kernel for operating system hardware support. In *Proceedings of the Fourth International Workshop on Object Orientations in Operating Systems*, pages 220–223, Lund, Sweden, August 1995.

[93] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[94] USENIX Association. *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, June 1990.

[95] USENIX Association. *1996 USENIX Annual Technical Conference*, San Diego, California, January 1996.

[96] Amin Vahdat, Douglas Ghormley, and Thomas Anderson. Efficient, portable, and robust extension of operating system functionality. Technical Report UCB CS-94-842, Computer Science Division, UC Berkeley, December 1994.

[97] Amin Vahdat, Douglas Ghormley, and Thomas Anderson. Efficient, portable, and robust extension of operating system functionality. In HOTOS 5 [42].

[98] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, February 1995.

[99] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In SOSP 15 [88], pages 40–53. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[100] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 256–266, Gold Coast, Australia, May 1992. ACM Press, New York. Published as ACM Computer Architecture News, SIGARCH, volume 20, number 2.

[101] G. Michael Vose. QuickBasic 4.0. *Byte*, 12(13):111–114, November 1987.

[102] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In SOSP 14 [87], pages 203–216. Published as ACM Operating Systems Review, SIGOPS, volume 27 number 5.

[103] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In PPOPP 5 [77], pages 217–226. Published as ACM SIGPLAN Notices, volume 30, number 8.

[104] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65. IEEE Computer Society Press, 1994. Available from: file://www.cs.sandia.gov/pub/sunmos/papers/hicss.ps.Z .

[105] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. *Scientific Programming*, 3:275–288, 1994.

[106] N. Wirth. Recollections about the development of Pascal. *SIGPLAN notices*, 28(3):333–342, March 1995.

[107] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Technical Conference Proceedings*, pages 153–165, San Diego, CA, Winter 1994. USENIX.